Unreliable Guide To Locking

Rusty Russell

rusty@rustcorp.com.au

Unreliable Guide To Locking
by Rusty Russell

Table of Contents

List of Tables

Chapter 1. Introduction

Welcome, to Rusty's Remarkably Unreliable Guide to Kernel Locking issues. This document describes the locking systems in the Linux Kernel in 2.6.

With the wide availability of HyperThreading, and *preemption* in the Linux Kernel, everyone hacking on the kernel needs to know the fundamentals of concurrency and locking for *SMP*.

Chapter 2. The Problem With Concurrency

(Skip this if you know what a Race Condition is).

In a normal program, you can increment a counter like so:

```
very_important_count++;
```

This is what they would expect to happen:

Table 2-1. Expected Results

Instance 1	Instance 2
read very_important_count (5)	
add 1 (6)	
write very_important_count (6)	
	read very_important_count (6)
	add 1 (7)
	write very_important_count (7)

This is what might happen:

Table 2-2. Possible Results

Instance 1	Instance 2
read very_important_count (5)	
	read very_important_count (5)
add 1 (6)	
	add 1 (6)
write very_important_count (6)	
	write very_important_count (6)

2.1. Race Conditions and Critical Regions

This overlap, where the result depends on the relative timing of multiple tasks, is called a *race condition*. The piece of code containing the concurrency issue is called a *critical region*. And especially since Linux starting running on SMP machines, they became one of the major issues in kernel design and implementation.

Preemption can have the same effect, even if there is only one CPU: by preempting one task during the

critical region, we have exactly the same race condition. In this case the thread which preempts might run the critical region itself.

The solution is to recognize when these simultaneous accesses occur, and use locks to make sure that only one instance can enter the critical region at any time. There are many friendly primitives in the Linux kernel to help you do this. And then there are the unfriendly primitives, but I'll pretend they don't exist.

Chapter 3. Locking in the Linux Kernel

If I could give you one piece of advice: never sleep with anyone crazier than yourself. But if I had to give you advice on locking: *keep it simple*.

Be reluctant to introduce new locks.

Strangely enough, this last one is the exact reverse of my advice when you *have* slept with someone crazier than yourself. And you should think about getting a big dog.

3.1. Two Main Types of Kernel Locks: Spinlocks and Mutexes

There are two main types of kernel locks. The fundamental type is the spinlock (`include/asm/spinlock.h`), which is a very simple single-holder lock: if you can't get the spinlock, you keep trying (spinning) until you can. Spinlocks are very small and fast, and can be used anywhere.

The second type is a mutex (`include/linux/mutex.h`): it is like a spinlock, but you may block holding a mutex. If you can't lock a mutex, your task will suspend itself, and be woken up when the mutex is released. This means the CPU can do something else while you are waiting. There are many cases when you simply can't sleep (see Chapter 10), and so have to use a spinlock instead.

Neither type of lock is recursive: see Section 8.1.

3.2. Locks and Uniprocessor Kernels

For kernels compiled without CONFIG_SMP, and without CONFIG_PREEMPT spinlocks do not exist at all. This is an excellent design decision: when no-one else can run at the same time, there is no reason to have a lock.

If the kernel is compiled without CONFIG_SMP, but CONFIG_PREEMPT is set, then spinlocks simply disable preemption, which is sufficient to prevent any races. For most purposes, we can think of preemption as equivalent to SMP, and not worry about it separately.

You should always test your locking code with CONFIG_SMP and CONFIG_PREEMPT enabled, even if you don't have an SMP test box, because it will still catch some kinds of locking bugs.

Mutexes still exist, because they are required for synchronization between *user contexts*, as we will see below.

3.3. Locking Only In User Context

If you have a data structure which is only ever accessed from user context, then you can use a simple mutex (`include/linux/mutex.h`) to protect it. This is the most trivial case: you initialize the mutex. Then you can call `mutex_lock_interruptible()` to grab the mutex, and `mutex_unlock()` to release it. There is also a `mutex_lock()`, which should be avoided, because it will not return if a signal is received.

Example: `net/netfilter/nf_sockopt.c` allows registration of new `setsockopt()` and `getsockopt()` calls, with `nf_register_sockopt()`. Registration and de-registration are only done on module load and unload (and boot time, where there is no concurrency), and the list of registrations is only consulted for an unknown `setsockopt()` or `getsockopt()` system call. The `nf_sockopt_mutex` is perfect to protect this, especially since the setsockopt and getsockopt calls may well sleep.

3.4. Locking Between User Context and Softirqs

If a *softirq* shares data with user context, you have two problems. Firstly, the current user context can be interrupted by a softirq, and secondly, the critical region could be entered from another CPU. This is where `spin_lock_bh()` (`include/linux/spinlock.h`) is used. It disables softirqs on that CPU, then grabs the lock. `spin_unlock_bh()` does the reverse. (The '_bh' suffix is a historical reference to "Bottom Halves", the old name for software interrupts. It should really be called spin_lock_softirq()' in a perfect world).

Note that you can also use `spin_lock_irq()` or `spin_lock_irqsave()` here, which stop hardware interrupts as well: see Chapter 4.

This works perfectly for *UP* as well: the spin lock vanishes, and this macro simply becomes `local_bh_disable()` (`include/linux/interrupt.h`), which protects you from the softirq being run.

3.5. Locking Between User Context and Tasklets

This is exactly the same as above, because *tasklets* are actually run from a softirq.

3.6. Locking Between User Context and Timers

This, too, is exactly the same as above, because *timers* are actually run from a softirq. From a locking point of view, tasklets and timers are identical.

3.7. Locking Between Tasklets/Timers

Sometimes a tasklet or timer might want to share data with another tasklet or timer.

3.7.1. The Same Tasklet/Timer

Since a tasklet is never run on two CPUs at once, you don't need to worry about your tasklet being reentrant (running twice at once), even on SMP.

3.7.2. Different Tasklets/Timers

If another tasklet/timer wants to share data with your tasklet or timer , you will both need to use `spin_lock()` and `spin_unlock()` calls. `spin_lock_bh()` is unnecessary here, as you are already in a tasklet, and none will be run on the same CPU.

3.8. Locking Between Softirqs

Often a softirq might want to share data with itself or a tasklet/timer.

3.8.1. The Same Softirq

The same softirq can run on the other CPUs: you can use a per-CPU array (see Section 9.3) for better performance. If you're going so far as to use a softirq, you probably care about scalable performance enough to justify the extra complexity.

You'll need to use `spin_lock()` and `spin_unlock()` for shared data.

3.8.2. Different Softirqs

You'll need to use `spin_lock()` and `spin_unlock()` for shared data, whether it be a timer, tasklet, different softirq or the same or another softirq: any of them could be running on a different CPU.

Chapter 4. Hard IRQ Context

Hardware interrupts usually communicate with a tasklet or softirq. Frequently this involves putting work in a queue, which the softirq will take out.

4.1. Locking Between Hard IRQ and Softirqs/Tasklets

If a hardware irq handler shares data with a softirq, you have two concerns. Firstly, the softirq processing can be interrupted by a hardware interrupt, and secondly, the critical region could be entered by a hardware interrupt on another CPU. This is where `spin_lock_irq()` is used. It is defined to disable interrupts on that cpu, then grab the lock. `spin_unlock_irq()` does the reverse.

The irq handler does not to use `spin_lock_irq()`, because the softirq cannot run while the irq handler is running: it can use `spin_lock()`, which is slightly faster. The only exception would be if a different hardware irq handler uses the same lock: `spin_lock_irq()` will stop that from interrupting us.

This works perfectly for UP as well: the spin lock vanishes, and this macro simply becomes `local_irq_disable()` (`include/asm/smp.h`), which protects you from the softirq/tasklet/BH being run.

`spin_lock_irqsave()` (`include/linux/spinlock.h`) is a variant which saves whether interrupts were on or off in a flags word, which is passed to `spin_unlock_irqrestore()`. This means that the same code can be used inside an hard irq handler (where interrupts are already off) and in softirqs (where the irq disabling is required).

Note that softirqs (and hence tasklets and timers) are run on return from hardware interrupts, so `spin_lock_irq()` also stops these. In that sense, `spin_lock_irqsave()` is the most general and powerful locking function.

4.2. Locking Between Two Hard IRQ Handlers

It is rare to have to share data between two IRQ handlers, but if you do, `spin_lock_irqsave()` should be used: it is architecture-specific whether all interrupts are disabled inside irq handlers themselves.

Chapter 5. Cheat Sheet For Locking

Pete Zaitcev gives the following summary:

- If you are in a process context (any syscall) and want to lock other process out, use a mutex. You can take a mutex and sleep (`copy_from_user*(` or `kmalloc(x,GFP_KERNEL)`).
- Otherwise (== data can be touched in an interrupt), use `spin_lock_irqsave()` and `spin_unlock_irqrestore()`.
- Avoid holding spinlock for more than 5 lines of code and across any function call (except accessors like `readb`).

5.1. Table of Minimum Requirements

The following table lists the *minimum* locking requirements between various contexts. In some cases, the same context can only be running on one CPU at a time, so no locking is required for that context (eg. a particular thread can only run on one CPU at a time, but if it needs shares data with another thread, locking is required).

Remember the advice above: you can always use `spin_lock_irqsave()`, which is a superset of all other spinlock primitives.

Table 5-1. Table of Locking Requirements

	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	SLIS	None								
Softirq A	SLI	SLI	SL							
Softirq B	SLI	SLI	SL	SL						
Tasklet A	SLI	SLI	SL	SL	None					

Tasklet B	SLI	SLI	SL	SL	SL	None				
Timer A	SLI	SLI	SL	SL	SL	SL	None			
Timer B	SLI	SLI	SL	SL	SL	SL	SL	None		
User Context A	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	None	
User Context B	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	MLI	None

Table 5-2. Legend for Locking Requirements Table

SLIS	spin_lock_irqsave
SLI	spin_lock_irq
SL	spin_lock
SLBH	spin_lock_bh
MLI	mutex_lock_interruptible

Chapter 6. The trylock Functions

There are functions that try to acquire a lock only once and immediately return a value telling about success or failure to acquire the lock. They can be used if you need no access to the data protected with the lock when some other thread is holding the lock. You should acquire the lock later if you then need access to the data protected with the lock.

`spin_trylock()` does not spin but returns non-zero if it acquires the spinlock on the first try or 0 if not. This function can be used in all contexts like `spin_lock`: you must have disabled the contexts that might interrupt you and acquire the spin lock.

`mutex_trylock()` does not suspend your task but returns non-zero if it could lock the mutex on the first try or 0 if not. This function cannot be safely used in hardware or software interrupt contexts despite not sleeping.

Chapter 7. Common Examples

Let's step through a simple example: a cache of number to name mappings. The cache keeps a count of how often each of the objects is used, and when it gets full, throws out the least used one.

7.1. All In User Context

For our first example, we assume that all operations are in user context (ie. from system calls), so we can sleep. This means we can use a mutex to protect the cache and all the objects within it. Here's the code:

```
#include <linux/list.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/mutex.h>
#include <asm/errno.h>

struct object
{
        struct list_head list;
        int id;
        char name[32];
        int popularity;
};

/* Protects the cache, cache_num, and the objects within it */
static DEFINE_MUTEX(cache_lock);
static LIST_HEAD(cache);
static unsigned int cache_num = 0;
#define MAX_CACHE_SIZE 10

/* Must be holding cache_lock */
static struct object *__cache_find(int id)
{
        struct object *i;

        list_for_each_entry(i, &cache, list)
                if (i->id == id) {
                        i->popularity++;
                        return i;
                }
        return NULL;
}

/* Must be holding cache_lock */
static void __cache_delete(struct object *obj)
{
        BUG_ON(!obj);
        list_del(&obj->list);
```

```
        kfree(obj);
        cache_num--;
}

/* Must be holding cache_lock */
static void __cache_add(struct object *obj)
{
        list_add(&obj->list, &cache);
        if (++cache_num > MAX_CACHE_SIZE) {
                struct object *i, *outcast = NULL;
                list_for_each_entry(i, &cache, list) {
                        if (!outcast || i->popularity < outcast->popularity)
                                outcast = i;
                }
                __cache_delete(outcast);
        }
}

int cache_add(int id, const char *name)
{
        struct object *obj;

        if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                return -ENOMEM;

        strlcpy(obj->name, name, sizeof(obj->name));
        obj->id = id;
        obj->popularity = 0;

        mutex_lock(&cache_lock);
        __cache_add(obj);
        mutex_unlock(&cache_lock);
        return 0;
}

void cache_delete(int id)
{
        mutex_lock(&cache_lock);
        __cache_delete(__cache_find(id));
        mutex_unlock(&cache_lock);
}

int cache_find(int id, char *name)
{
        struct object *obj;
        int ret = -ENOENT;

        mutex_lock(&cache_lock);
        obj = __cache_find(id);
        if (obj) {
                ret = 0;
                strcpy(name, obj->name);
        }
```

```
        mutex_unlock(&cache_lock);
        return ret;
}
```

Note that we always make sure we have the cache_lock when we add, delete, or look up the cache: both the cache infrastructure itself and the contents of the objects are protected by the lock. In this case it's easy, since we copy the data for the user, and never let them access the objects directly.

There is a slight (and common) optimization here: in `cache_add` we set up the fields of the object before grabbing the lock. This is safe, as no-one else can access it until we put it in cache.

7.2. Accessing From Interrupt Context

Now consider the case where `cache_find` can be called from interrupt context: either a hardware interrupt or a softirq. An example would be a timer which deletes object from the cache.

The change is shown below, in standard patch format: the - are lines which are taken away, and the + are lines which are added.

```
--- cache.c.usercontext 2003-12-09 13:58:54.000000000 +1100
+++ cache.c.interrupt 2003-12-09 14:07:49.000000000 +1100
@@ -12,7 +12,7 @@
         int popularity;
 };

-static DEFINE_MUTEX(cache_lock);
+static DEFINE_SPINLOCK(cache_lock);
 static LIST_HEAD(cache);
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10
@@ -55,6 +55,7 @@
 int cache_add(int id, const char *name)
 {
         struct object *obj;
+        unsigned long flags;

         if ((obj = kmalloc(sizeof(*obj), GFP_KERNEL)) == NULL)
                 return -ENOMEM;
@@ -63,30 +64,33 @@
         obj->id = id;
         obj->popularity = 0;

-        mutex_lock(&cache_lock);
+        spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
```

```
         return 0;
 }

 void cache_delete(int id)
 {
-        mutex_lock(&cache_lock);
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
         __cache_delete(__cache_find(id));
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
 }

 int cache_find(int id, char *name)
 {
         struct object *obj;
         int ret = -ENOENT;
+        unsigned long flags;

-        mutex_lock(&cache_lock);
+        spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
         if (obj) {
                 ret = 0;
                 strcpy(name, obj->name);
         }
-        mutex_unlock(&cache_lock);
+        spin_unlock_irqrestore(&cache_lock, flags);
         return ret;
 }
```

Note that the `spin_lock_irqsave` will turn off interrupts if they are on, otherwise does nothing (if we are already in an interrupt handler), hence these functions are safe to call from any context.

Unfortunately, `cache_add` calls `kmalloc` with the GFP_KERNEL flag, which is only legal in user context. I have assumed that `cache_add` is still only called in user context, otherwise this should become a parameter to `cache_add`.

7.3. Exposing Objects Outside This File

If our objects contained more information, it might not be sufficient to copy the information in and out: other parts of the code might want to keep pointers to these objects, for example, rather than looking up the id every time. This produces two problems.

The first problem is that we use the cache_lock to protect objects: we'd need to make this non-static so the rest of the code can use it. This makes locking trickier, as it is no longer all in one place.

The second problem is the lifetime problem: if another structure keeps a pointer to an object, it presumably expects that pointer to remain valid. Unfortunately, this is only guaranteed while you hold the lock, otherwise someone might call `cache_delete` and even worse, add another object, re-using the same address.

As there is only one lock, you can't hold it forever: no-one else would get any work done.

The solution to this problem is to use a reference count: everyone who has a pointer to the object increases it when they first get the object, and drops the reference count when they're finished with it. Whoever drops it to zero knows it is unused, and can actually delete it.

Here is the code:

```
--- cache.c.interrupt 2003-12-09 14:25:43.000000000 +1100
+++ cache.c.refcnt 2003-12-09 14:33:05.000000000 +1100
@@ -7,6 +7,7 @@
 struct object
 {
         struct list_head list;
+        unsigned int refcnt;
         int id;
         char name[32];
         int popularity;
@@ -17,6 +18,35 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

+static void __object_put(struct object *obj)
+{
+        if (--obj->refcnt == 0)
+                kfree(obj);
+}
+
+static void __object_get(struct object *obj)
+{
+        obj->refcnt++;
+}
+
+void object_put(struct object *obj)
+{
+        unsigned long flags;
+
+        spin_lock_irqsave(&cache_lock, flags);
+        __object_put(obj);
+        spin_unlock_irqrestore(&cache_lock, flags);
+}
+
+void object_get(struct object *obj)
+{
+        unsigned long flags;
```

```
+
+        spin_lock_irqsave(&cache_lock, flags);
+        __object_get(obj);
+        spin_unlock_irqrestore(&cache_lock, flags);
+}
+
 /* Must be holding cache_lock */
 static struct object *__cache_find(int id)
 {
@@ -35,6 +65,7 @@
 {
         BUG_ON(!obj);
         list_del(&obj->list);
+        __object_put(obj);
         cache_num--;
 }

@@ -63,6 +94,7 @@
         strlcpy(obj->name, name, sizeof(obj->name));
         obj->id = id;
         obj->popularity = 0;
+        obj->refcnt = 1; /* The cache holds a reference */

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
@@ -79,18 +111,15 @@
         spin_unlock_irqrestore(&cache_lock, flags);
 }

-int cache_find(int id, char *name)
+struct object *cache_find(int id)
 {
         struct object *obj;
-        int ret = -ENOENT;
         unsigned long flags;

         spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
-        if (obj) {
-                ret = 0;
-                strcpy(name, obj->name);
-        }
+        if (obj)
+                __object_get(obj);
         spin_unlock_irqrestore(&cache_lock, flags);
-        return ret;
+        return obj;
 }
```

We encapsulate the reference counting in the standard 'get' and 'put' functions. Now we can return the object itself from `cache_find` which has the advantage that the user can now sleep holding the object (eg. to `copy_to_user` to name to userspace).

The other point to note is that I said a reference should be held for every pointer to the object: thus the reference count is 1 when first inserted into the cache. In some versions the framework does not hold a reference count, but they are more complicated.

7.3.1. Using Atomic Operations For The Reference Count

In practice, atomic_t would usually be used for *`refcnt`*. There are a number of atomic operations defined in `include/asm/atomic.h`: these are guaranteed to be seen atomically from all CPUs in the system, so no lock is required. In this case, it is simpler than using spinlocks, although for anything non-trivial using spinlocks is clearer. The `atomic_inc` and `atomic_dec_and_test` are used instead of the standard increment and decrement operators, and the lock is no longer used to protect the reference count itself.

```
--- cache.c.refcnt 2003-12-09 15:00:35.000000000 +1100
+++ cache.c.refcnt-atomic 2003-12-11 15:49:42.000000000 +1100
@@ -7,7 +7,7 @@
 struct object
 {
         struct list_head list;
-        unsigned int refcnt;
+        atomic_t refcnt;
         int id;
         char name[32];
         int popularity;
@@ -18,33 +18,15 @@
 static unsigned int cache_num = 0;
 #define MAX_CACHE_SIZE 10

-static void __object_put(struct object *obj)
-{
-        if (--obj->refcnt == 0)
-                kfree(obj);
-}
-
-static void __object_get(struct object *obj)
-{
-        obj->refcnt++;
-}
-
 void object_put(struct object *obj)
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&cache_lock, flags);
-        __object_put(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        if (atomic_dec_and_test(&obj->refcnt))
+                kfree(obj);
 }

 void object_get(struct object *obj)
```

```
 {
-        unsigned long flags;
-
-        spin_lock_irqsave(&cache_lock, flags);
-        __object_get(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        atomic_inc(&obj->refcnt);
 }

 /* Must be holding cache_lock */
@@ -65,7 +47,7 @@
 {
         BUG_ON(!obj);
         list_del(&obj->list);
-        __object_put(obj);
+        object_put(obj);
         cache_num--;
 }

@@ -94,7 +76,7 @@
         strlcpy(obj->name, name, sizeof(obj->name));
         obj->id = id;
         obj->popularity = 0;
-        obj->refcnt = 1; /* The cache holds a reference */
+        atomic_set(&obj->refcnt, 1); /* The cache holds a reference */

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
@@ -119,7 +101,7 @@
         spin_lock_irqsave(&cache_lock, flags);
         obj = __cache_find(id);
         if (obj)
-                __object_get(obj);
+                object_get(obj);
         spin_unlock_irqrestore(&cache_lock, flags);
         return obj;
 }
```

7.4. Protecting The Objects Themselves

In these examples, we assumed that the objects (except the reference counts) never changed once they are created. If we wanted to allow the name to change, there are three possibilities:

- You can make cache_lock non-static, and tell people to grab that lock before changing the name in any object.
- You can provide a `cache_obj_rename` which grabs this lock and changes the name for the caller, and tell everyone to use that function.

- You can make the cache_lock protect only the cache itself, and use another lock to protect the name.

Theoretically, you can make the locks as fine-grained as one lock for every field, for every object. In practice, the most common variants are:

- One lock which protects the infrastructure (the cache list in this example) and all the objects. This is what we have done so far.
- One lock which protects the infrastructure (including the list pointers inside the objects), and one lock inside the object which protects the rest of that object.
- Multiple locks to protect the infrastructure (eg. one lock per hash chain), possibly with a separate per-object lock.

Here is the "lock-per-object" implementation:

```
--- cache.c.refcnt-atomic 2003-12-11 15:50:54.000000000 +1100
+++ cache.c.perobjectlock 2003-12-11 17:15:03.000000000 +1100
@@ -6,11 +6,17 @@

 struct object
 {
+        /* These two protected by cache_lock. */
         struct list_head list;
+        int popularity;
+
         atomic_t refcnt;
+
+        /* Doesn't change once created. */
         int id;
+
+        spinlock_t lock; /* Protects the name */
         char name[32];
-        int popularity;
 };

 static DEFINE_SPINLOCK(cache_lock);
@@ -77,6 +84,7 @@
         obj->id = id;
         obj->popularity = 0;
         atomic_set(&obj->refcnt, 1); /* The cache holds a reference */
+        spin_lock_init(&obj->lock);

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
```

Note that I decide that the *`popularity`* count should be protected by the cache_lock rather than the per-object lock: this is because it (like the struct list_head inside the object) is logically part of the infrastructure. This way, I don't need to grab the lock of every object in `__cache_add` when seeking the least popular.

I also decided that the *`id`* member is unchangeable, so I don't need to grab each object lock in `__cache_find()` to examine the *`id`*: the object lock is only used by a caller who wants to read or write the *`name`* field.

Note also that I added a comment describing what data was protected by which locks. This is extremely important, as it describes the runtime behavior of the code, and can be hard to gain from just reading. And as Alan Cox says, “Lock data, not code”.

Chapter 8. Common Problems

8.1. Deadlock: Simple and Advanced

There is a coding bug where a piece of code tries to grab a spinlock twice: it will spin forever, waiting for the lock to be released (spinlocks, rwlocks and mutexes are not recursive in Linux). This is trivial to diagnose: not a stay-up-five-nights-talk-to-fluffy-code-bunnies kind of problem.

For a slightly more complex case, imagine you have a region shared by a softirq and user context. If you use a `spin_lock()` call to protect it, it is possible that the user context will be interrupted by the softirq while it holds the lock, and the softirq will then spin forever trying to get the same lock.

Both of these are called deadlock, and as shown above, it can occur even with a single CPU (although not on UP compiles, since spinlocks vanish on kernel compiles with CONFIG_SMP=n. You'll still get data corruption in the second example).

This complete lockup is easy to diagnose: on SMP boxes the watchdog timer or compiling with DEBUG_SPINLOCK set (`include/linux/spinlock.h`) will show this up immediately when it happens.

A more complex problem is the so-called 'deadly embrace', involving two or more locks. Say you have a hash table: each entry in the table is a spinlock, and a chain of hashed objects. Inside a softirq handler, you sometimes want to alter an object from one place in the hash to another: you grab the spinlock of the old hash chain and the spinlock of the new hash chain, and delete the object from the old one, and insert it in the new one.

There are two problems here. First, if your code ever tries to move the object to the same chain, it will deadlock with itself as it tries to lock it twice. Secondly, if the same softirq on another CPU is trying to move another object in the reverse direction, the following could happen:

Table 8-1. Consequences

CPU 1	CPU 2
Grab lock A -> OK	Grab lock B -> OK
Grab lock B -> spin	Grab lock A -> spin

The two CPUs will spin forever, waiting for the other to give up their lock. It will look, smell, and feel like a crash.

8.2. Preventing Deadlock

Textbooks will tell you that if you always lock in the same order, you will never get this kind of deadlock. Practice will tell you that this approach doesn't scale: when I create a new lock, I don't understand enough of the kernel to figure out where in the 5000 lock hierarchy it will fit.

The best locks are encapsulated: they never get exposed in headers, and are never held around calls to non-trivial functions outside the same file. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

A classic problem here is when you provide callbacks or hooks: if you call these with the lock held, you risk simple deadlock, or a deadly embrace (who knows what the callback will do?). Remember, the other programmers are out to get you, so don't do this.

8.2.1. Overzealous Prevention Of Deadlocks

Deadlocks are problematic, but not as bad as data corruption. Code which grabs a read lock, searches a list, fails to find what it wants, drops the read lock, grabs a write lock and inserts the object has a race condition.

If you don't see why, please stay the fuck away from my code.

8.3. Racing Timers: A Kernel Pastime

Timers can produce their own special problems with races. Consider a collection of objects (list, hash, etc) where each object has a timer which is due to destroy it.

If you want to destroy the entire collection (say on module removal), you might do the following:

```
/* THIS CODE BAD BAD BAD BAD: IF IT WAS ANY WORSE IT WOULD USE
   HUNGARIAN NOTATION */
spin_lock_bh(&list_lock);

while (list) {
        struct foo *next = list->next;
        del_timer(&list->timer);
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

Sooner or later, this will crash on SMP, because a timer can have just gone off before the `spin_lock_bh()`, and it will only get the lock after we `spin_unlock_bh()`, and then try to free the element (which has already been freed!).

This can be avoided by checking the result of `del_timer()`: if it returns 1, the timer has been deleted. If 0, it means (in this case) that it is currently running, so we can do:

```
retry:
        spin_lock_bh(&list_lock);

        while (list) {
                struct foo *next = list->next;
                if (!del_timer(&list->timer)) {
                        /* Give timer a chance to delete this */
                        spin_unlock_bh(&list_lock);
                        goto retry;
                }
                kfree(list);
                list = next;
        }

        spin_unlock_bh(&list_lock);
```

Another common problem is deleting timers which restart themselves (by calling `add_timer()` at the end of their timer function). Because this is a fairly common case which is prone to races, you should use `del_timer_sync()` (`include/linux/timer.h`) to handle this case. It returns the number of times the timer had to be deleted before we finally stopped it from adding itself back in.

Chapter 9. Locking Speed

There are three main things to worry about when considering speed of some code which does locking. First is concurrency: how many things are going to be waiting while someone else is holding a lock. Second is the time taken to actually acquire and release an uncontended lock. Third is using fewer, or smarter locks. I'm assuming that the lock is used fairly often: otherwise, you wouldn't be concerned about efficiency.

Concurrency depends on how long the lock is usually held: you should hold the lock for as long as needed, but no longer. In the cache example, we always create the object without the lock held, and then grab the lock only when we are ready to insert it in the list.

Acquisition times depend on how much damage the lock operations do to the pipeline (pipeline stalls) and how likely it is that this CPU was the last one to grab the lock (ie. is the lock cache-hot for this CPU): on a machine with more CPUs, this likelihood drops fast. Consider a 700MHz Intel Pentium III: an instruction takes about 0.7ns, an atomic increment takes about 58ns, a lock which is cache-hot on this CPU takes 160ns, and a cacheline transfer from another CPU takes an additional 170 to 360ns. (These figures from Paul McKenney's Linux Journal RCU article (http://www.linuxjournal.com/article.php?sid=6993)).

These two aims conflict: holding a lock for a short time might be done by splitting locks into parts (such as in our final per-object-lock example), but this increases the number of lock acquisitions, and the results are often slower than having a single lock. This is another reason to advocate locking simplicity.

The third concern is addressed below: there are some methods to reduce the amount of locking which needs to be done.

9.1. Read/Write Lock Variants

Both spinlocks and mutexes have read/write variants: rwlock_t and struct rw_semaphore. These divide users into two classes: the readers and the writers. If you are only reading the data, you can get a read lock, but to write to the data you need the write lock. Many people can hold a read lock, but a writer must be sole holder.

If your code divides neatly along reader/writer lines (as our cache code does), and the lock is held by readers for significant lengths of time, using these locks can help. They are slightly slower than the normal locks though, so in practice rwlock_t is not usually worthwhile.

9.2. Avoiding Locks: Read Copy Update

There is a special method of read/write locking called Read Copy Update. Using RCU, the readers can avoid taking a lock altogether: as we expect our cache to be read more often than updated (otherwise the cache is a waste of time), it is a candidate for this optimization.

How do we get rid of read locks? Getting rid of read locks means that writers may be changing the list underneath the readers. That is actually quite simple: we can read a linked list while an element is being added if the writer adds the element very carefully. For example, adding new to a single linked list called list:

```
new->next = list->next;
wmb();
list->next = new;
```

The `wmb()` is a write memory barrier. It ensures that the first operation (setting the new element's next pointer) is complete and will be seen by all CPUs, before the second operation is (putting the new element into the list). This is important, since modern compilers and modern CPUs can both reorder instructions unless told otherwise: we want a reader to either not see the new element at all, or see the new element with the next pointer correctly pointing at the rest of the list.

Fortunately, there is a function to do this for standard struct list_head lists: `list_add_rcu()` (`include/linux/list.h`).

Removing an element from the list is even simpler: we replace the pointer to the old element with a pointer to its successor, and readers will either see it, or skip over it.

```
list->next = old->next;
```

There is `list_del_rcu()` (`include/linux/list.h`) which does this (the normal version poisons the old object, which we don't want).

The reader must also be careful: some CPUs can look through the next pointer to start reading the contents of the next element early, but don't realize that the pre-fetched contents is wrong when the next pointer changes underneath them. Once again, there is a `list_for_each_entry_rcu()` (`include/linux/list.h`) to help you. Of course, writers can just use `list_for_each_entry()`, since there cannot be two simultaneous writers.

Our final dilemma is this: when can we actually destroy the removed element? Remember, a reader might be stepping through this element in the list right now: if we free this element and the next pointer changes, the reader will jump off into garbage and crash. We need to wait until we know that all the

readers who were traversing the list when we deleted the element are finished. We use `call_rcu()` to register a callback which will actually destroy the object once the readers are finished.

But how does Read Copy Update know when the readers are finished? The method is this: firstly, the readers always traverse the list inside `rcu_read_lock()`/`rcu_read_unlock()` pairs: these simply disable preemption so the reader won't go to sleep while reading the list.

RCU then waits until every other CPU has slept at least once: since readers cannot sleep, we know that any readers which were traversing the list during the deletion are finished, and the callback is triggered. The real Read Copy Update code is a little more optimized than this, but this is the fundamental idea.

```
--- cache.c.perobjectlock 2003-12-11 17:15:03.000000000 +1100
+++ cache.c.rcupdate 2003-12-11 17:55:14.000000000 +1100
@@ -1,15 +1,18 @@
 #include <linux/list.h>
 #include <linux/slab.h>
 #include <linux/string.h>
+#include <linux/rcupdate.h>
 #include <linux/mutex.h>
 #include <asm/errno.h>

 struct object
 {
-        /* These two protected by cache_lock. */
+        /* This is protected by RCU */
         struct list_head list;
         int popularity;

+        struct rcu_head rcu;
+
         atomic_t refcnt;

         /* Doesn't change once created. */
@@ -40,7 +43,7 @@
 {
         struct object *i;

-        list_for_each_entry(i, &cache, list) {
+        list_for_each_entry_rcu(i, &cache, list) {
                 if (i->id == id) {
                         i->popularity++;
                         return i;
@@ -49,19 +52,25 @@
         return NULL;
 }

+/* Final discard done once we know no readers are looking. */
+static void cache_delete_rcu(void *arg)
+{
+        object_put(arg);
+}
```

```
+
 /* Must be holding cache_lock */
 static void __cache_delete(struct object *obj)
 {
         BUG_ON(!obj);
-        list_del(&obj->list);
-        object_put(obj);
+        list_del_rcu(&obj->list);
         cache_num--;
+        call_rcu(&obj->rcu, cache_delete_rcu, obj);
 }

 /* Must be holding cache_lock */
 static void __cache_add(struct object *obj)
 {
-        list_add(&obj->list, &cache);
+        list_add_rcu(&obj->list, &cache);
         if (++cache_num > MAX_CACHE_SIZE) {
                 struct object *i, *outcast = NULL;
                 list_for_each_entry(i, &cache, list) {
@@ -85,6 +94,7 @@
         obj->popularity = 0;
         atomic_set(&obj->refcnt, 1); /* The cache holds a reference */
         spin_lock_init(&obj->lock);
+        INIT_RCU_HEAD(&obj->rcu);

         spin_lock_irqsave(&cache_lock, flags);
         __cache_add(obj);
@@ -104,12 +114,11 @@
 struct object *cache_find(int id)
 {
         struct object *obj;
-        unsigned long flags;

-        spin_lock_irqsave(&cache_lock, flags);
+        rcu_read_lock();
         obj = __cache_find(id);
         if (obj)
                 object_get(obj);
-        spin_unlock_irqrestore(&cache_lock, flags);
+        rcu_read_unlock();
         return obj;
 }
```

Note that the reader will alter the *popularity* member in `__cache_find()`, and now it doesn't hold a lock. One solution would be to make it an atomic_t, but for this usage, we don't really care about races: an approximate result is good enough, so I didn't change it.

The result is that `cache_find()` requires no synchronization with any other functions, so is almost as fast on SMP as it would be on UP.

There is a furthur optimization possible here: remember our original cache code, where there were no reference counts and the caller simply held the lock whenever using the object? This is still possible: if you hold the lock, noone can delete the object, so you don't need to get and put the reference count.

Now, because the 'read lock' in RCU is simply disabling preemption, a caller which always has preemption disabled between calling `cache_find()` and `object_put()` does not need to actually get and put the reference count: we could expose `__cache_find()` by making it non-static, and such callers could simply call that.

The benefit here is that the reference count is not written to: the object is not altered in any way, which is much faster on SMP machines due to caching.

9.3. Per-CPU Data

Another technique for avoiding locking which is used fairly widely is to duplicate information for each CPU. For example, if you wanted to keep a count of a common condition, you could use a spin lock and a single counter. Nice and simple.

If that was too slow (it's usually not, but if you've got a really big machine to test on and can show that it is), you could instead use a counter for each CPU, then none of them need an exclusive lock. See `DEFINE_PER_CPU()`, `get_cpu_var()` and `put_cpu_var()` (`include/linux/percpu.h`).

Of particular use for simple per-cpu counters is the local_t type, and the `cpu_local_inc()` and related functions, which are more efficient than simple code on some architectures (`include/asm/local.h`).

Note that there is no simple, reliable way of getting an exact value of such a counter, without introducing more locks. This is not a problem for some uses.

9.4. Data Which Mostly Used By An IRQ Handler

If data is always accessed from within the same IRQ handler, you don't need a lock at all: the kernel already guarantees that the irq handler will not run simultaneously on multiple CPUs.

Manfred Spraul points out that you can still do this, even if the data is very occasionally accessed in user context or softirqs/tasklets. The irq handler doesn't use a lock, and all other accesses are done as so:

```
spin_lock(&lock);
disable_irq(irq);
...
enable_irq(irq);
spin_unlock(&lock);
```

The `disable_irq()` prevents the irq handler from running (and waits for it to finish if it's currently running on other CPUs). The spinlock prevents any other accesses happening at the same time. Naturally, this is slower than just a `spin_lock_irq()` call, so it only makes sense if this type of access happens extremely rarely.

Chapter 10. What Functions Are Safe To Call From Interrupts?

Many functions in the kernel sleep (ie. call schedule()) directly or indirectly: you can never call them while holding a spinlock, or with preemption disabled. This also means you need to be in user context: calling them from an interrupt is illegal.

10.1. Some Functions Which Sleep

The most common ones are listed below, but you usually have to read the code to find out if other calls are safe. If everyone else who calls it can sleep, you probably need to be able to sleep, too. In particular, registration and deregistration functions usually expect to be called from user context, and can sleep.

- Accesses to *userspace*:
  - `copy_from_user()`
  - `copy_to_user()`
  - `get_user()`
  - `put_user()`
- `kmalloc(GFP_KERNEL)`
- `mutex_lock_interruptible()` and `mutex_lock()`

  There is a `mutex_trylock()` which can be used inside interrupt context, as it will not sleep. `mutex_unlock()` will also never sleep.

10.2. Some Functions Which Don't Sleep

Some functions are safe to call from any context, or holding almost any lock.

- `printk()`
- `kfree()`
- `add_timer()` and `del_timer()`

Chapter 11. Further reading

- `Documentation/spinlocks.txt`: Linus Torvalds' spinlocking tutorial in the kernel sources.
- Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers:

  Curt Schimmel's very good introduction to kernel level locking (not written for Linux, but nearly everything applies). The book is expensive, but really worth every penny to understand SMP locking. [ISBN: 0201633388]

Chapter 12. Thanks

Thanks to Telsa Gwynne for DocBooking, neatening and adding style.

Thanks to Martin Pool, Philipp Rumpf, Stephen Rothwell, Paul Mackerras, Ruedi Aschwanden, Alan Cox, Manfred Spraul, Tim Waugh, Pete Zaitcev, James Morris, Robert Love, Paul McKenney, John Ashby for proofreading, correcting, flaming, commenting.

Thanks to the cabal for having no influence on this document.

Glossary

preemption

Prior to 2.5, or when CONFIG_PREEMPT is unset, processes in user context inside the kernel would not preempt each other (ie. you had that CPU until you gave it up, except for interrupts). With the addition of CONFIG_PREEMPT in 2.5.4, this changed: when in user context, higher priority tasks can "cut in": spinlocks were changed to disable preemption, even on UP.

bh

Bottom Half: for historical reasons, functions with '_bh' in them often now refer to any software interrupt, e.g. `spin_lock_bh()` blocks any software interrupt on the current CPU. Bottom halves are deprecated, and will eventually be replaced by tasklets. Only one bottom half will be running at any time.

Hardware Interrupt / Hardware IRQ

Hardware interrupt request. `in_irq()` returns true in a hardware interrupt handler.

Interrupt Context

Not user context: processing a hardware irq or software irq. Indicated by the `in_interrupt()` macro returning true.

SMP

Symmetric Multi-Processor: kernels compiled for multiple-CPU machines. (CONFIG_SMP=y).

Software Interrupt / softirq

Software interrupt handler. `in_irq()` returns false; `in_softirq()` returns true. Tasklets and softirqs both fall into the category of 'software interrupts'.

Strictly speaking a softirq is one of up to 32 enumerated software interrupts which can run on multiple CPUs at once. Sometimes used to refer to tasklets as well (ie. all software interrupts).

tasklet

A dynamically-registrable software interrupt, which is guaranteed to only run on one CPU at a time.

timer

A dynamically-registrable software interrupt, which is run at (or close to) a given time. When running, it is just like a tasklet (in fact, they are called from the TIMER_SOFTIRQ).

UP

Uni-Processor: Non-SMP. (CONFIG_SMP=n).

User Context

The kernel executing on behalf of a particular process (ie. a system call or trap) or kernel thread. You can tell which process with the current macro.) Not to be confused with userspace. Can be interrupted by software or hardware interrupts.

Userspace

A process executing its own code outside the kernel.